The upgradable proxy smart contract pattern is a simple and effective way to create contracts that can be upgraded, changed in time.

It allows developers to modify the functionality of a smart contract whilst preserving the contract address and storage.

But aren't smart contracts supposed to be immutable?

Upgradable Proxy patterns split the functionality (or implementation logic) and storage into separate contracts enabling the logic to be upgraded while maintaining immutability.

Sounds confusing? Let's dive into it.

Do we need upgradable smart contracts?

Smart contracts are immutable, meaning the business logic cannot be updated once deployed, ensuring trustlessness, decentralization, and security.

Although immutability is crucial for the integrity of smart contracts, it can pose challenges in situations where developers need to address vulnerabilities and modify the logic. Fortunately, upgradable smart contract patterns have been developed to address this issue.

One way to modify the contract functionality of a smart contract while preserving its state is by using proxy contracts. Proxy patterns allow developers to modify the smart contract functionality while maintaining their immutability by separating the business logic into different smart contracts. By using proxy contracts, the logic that is executed when users interact with a smart contract can be modified.

What is a Proxy smart contract?

The proxy pattern is a smart contract pattern that splits the business logic into at least two separate contracts:

1. A contract containing the data storage. This contract is known as a proxy contract

.

1. A contract containing the business logic. This contract is known as an implementation contract

.

Users interact directly with the proxy contract which forwards requests (using the delegatecall

function) to the implementation contract which dictates how the function executes and how the proxy's storage should be modified.

After the call is forwarded to the implementation contract, the data is retrieved by the proxy and returned to the user.

When a contract is upgraded, the implementation contract is re-deployed, containing the modified implementation logic. The proxy contract is subsequently updated using an "update" function call to re-route calls to the new implementation.

This way, the contract storage and address remain immutable, all while enabling the underlying logic to be updated or bugs to be fixed.

How do simple (delegate) upgradable Proxy smart contracts work?

Simple Proxy patterns

rely on low-level delegatecalls

. delegatecall

is an Ethereum opcode that allows a contract (contract A) to call another contract (contract B) in the context of contract A. This means that the storage of contract A is modified when a function on contract B is called and msg.sender

and msg.value

will be preserved. This means that the function call to contract B will look as if it was a call to a function on contract A.

The proxy contract invokes a delegatecall

whenever a user calls a function via a custom fallback

function. A fallback function is a function that executes when thefunction signature does not match any functions specified in the contract. This allows the contract to respond to arbitrary Ethereum transactions. The custom fallback function initiates the delegatecall

to reroute the user's call to the implementation contract where the function logic is held.

When the logic needs to be updated, a new implementation contract is deployed and the proxy contract is updated, by calling an upgrade function such as upgradeTo(address newImplementation)

to point to the new implementation address.

— For more information on how the proxy pattern works, please refer to theOpenZeppelin Proxy Patterns article.

What are Initializers?

As with constructors, initializers are used for initializing state in proxy smart patterns.

If a constructor was called on the implementation contract, it would affect only the implementation contract's storage rather than the proxy's. Therefore, initializer functions, implemented on the implementation contract and executed via a delegatecall

originating from the proxy contract, are used to initialize the state in the proxy's storage and align with the logic of the implementation contract.

- Instead of being executed once at deployment time, as with constructors, initializers are used and are called manually post-deployment.
- Unlike constructors, inheritance is not handled automatically, as an initializer is just a regular function. If parent contracts of the implementation contract have constructors, these must be called in the initializer function.
- Initializers must be called only once. This is usually enforced using a boolean to prevent security vulnerabilities. One way to ensure the initializer can only be called once is by using the OpenZeppelin Initializable contract.

- Initializers must be carefully constructed to only be callable as intended to maintain smart contract security when implementing proxy smart contract patterns.

— For more information on initializers, refer to theOpenZeppelin documentation on writing upgradable contracts.

The problem with the Simple Delegate Proxy Pattern

Simple proxy patterns have two major issues:

- Storage collisions
- Function selector collisions

Let's go through these now.

Storage collisions

The order of variable declarations in the Solidity code determines thestorage layout of a smart contract. When using upgradable smart contracts, this can become an issue due to the order of declarations differing between:

- The proxy and implementation contracts
- Different implementation contract versions

This can lead to issues when the data is read and modified leading to security vulnerabilities and bugs in the code.

Storage collisions between the Proxy and implementation contracts

The proxy pattern requires both the proxy and the implementation contracts to share the same storage layouts to prevent variables from being incorrectly overwritten or read.

Since the proxy contract needs to store the address of the implementation contract but the implementation contract does not, a common cause of storage collision is the implementation contract overwriting the implementation address slot on the proxy contract. To mitigate this problem, the ERC-1967 Proxy Storage Slots standard was created which defined a specific storage slot to store the implementation contract address.

Storage collisions between implementation versions

If the storage variable declaration order is changed between versions, the values will be overwritten or read incorrectly.

If storage slot 0 refers to “variable a” and slot 1 refers to “variable b” in the old implementation contract but in the upgraded, new implementation, slot 0 refers to “variable b” and slot 1 refers to “variable b”, when storage slot 0 is read or modified, “variable a” will be accessed instead.

In this example, if we perform the upgrade so that Contract B is the new implementation and then access “variable a” using its original storage slot 0, it would mistakenly interact with “variable b”.

When creating new implementation contracts to perform an upgrade, it is therefore crucial to ensure that the order of storage variables remains consistent.

Function collisions

Functions in a smart contract are selected using their function selector. The function selector is the first 4 bytes of the keccak256 hash of the function signature, defined using the function name, input types, state visibility, modifiers, and return type. When function selectors clash, e.g. functions with the same selector exist in both the proxy and implementation contract, when a user calls a function, the proxy does not know whether to call its own function or perform a delegatecall

. This can occur if functions on the implementation and proxy contracts have the same signature e.g. the implementation also has an update(address)

function, leading to ambiguity as to whether to perform the delegatecall

or not.

In another example explained in depth in thisOpenZeppelin article by Tincho, an attacker creates a proxy that points to a token implementation contract:

In this case, collate_propagate_storage

has a function signature of 0x42966c68

. Incidentally, the function signature of the burn

function on the implementation also has a function signature of 0x42966c68

. Therefore, if a user called burn

through the proxy, intending to burn 1 token for example, instead they would accidentally transfer 1000 tokens to the attacker!

To mitigate this, OpenZeppelin developed a few proxy patterns:

- The Transparent Proxy Pattern
- The UUPS pattern
- The Beacon Broxy Pattern

What is the Transparent Proxy Pattern?

Transparent Proxy Patterns work by interpreting message calls based on their origin, i.e. using the value of msg.sender

and providing different execution paths based on whether the address is an admin or non-admin user. The proxy contract retrieves the sender and depending on the value, determines whether to re-route the call to the implementation contract:

- If the msg.sender

is the user, i.e. is not an admin, the calls are delegated

to the implementation contract. This is the usual proxy pattern behavior.

- If the msg.sender

is the admin of the proxy, the proxy will invoke its administrative functions on the proxy, and calls are not delegated

. Admin actions are instigated via a ProxyAdmin

contract with the owner

role that will forward calls to the proxy contract. This is to avoid unwanted reverts when a user, who is also the admin, interacts with the contract.

This means that admins can only interact with the proxy contract functions and non-admins can only interact with the implementation contract functions, thus removing the ambiguity.

Transparent Proxy downsides

- Non-admin users are no longer able to access read methods

, such as variable getters e.g. for the implementation address, on the proxy contract.ERC-1967 solved this issue by providing a specific storage slot for the implementation contract and admin addresses meaning that users can view these values.

- They cost more gas to interact with

since the proxy needs to determine whether to reroute the call and therefore needs to load the admin address on every call.

- Deployment is more costly

to deploy due to this extra logic.

This led to the development of the UUPS proxy pattern. Let's go through this one now.

For more information, refer to theOpenZeppelin transparent proxy pattern article.

What is the Universal Upgrade Proxy Standard (UUPS)?

As with simple and transparent proxy patterns, UUPS proxies, proposed inEIP-1822, use the delegatecall

function. The difference is that UUPS proxies use the implementation contract, rather than the proxy contract to manage upgrades.

The proxy contract becomes a simple forwarding contract to forward all calls to the implementation contract via a delegatecall

, removing the gas overhead needed for loading the admin address and checking the calling address on every call as in the transparent proxy pattern.

How do UUPS Proxies work?

The implementation contract inherits a "proxiable contract"

, such as OpenZeppelin's abstract UUPSUpgradable contract (which provides one function _authorizeUpgrade(address newImplementation)

undefined for the developer to implement custom authorization logic), which provides the upgrade functionality. This means that calls to upgrade are the only time that the admin address needs to be loaded, reducing gas costs.

Each new implementation contract will inherit a "proxiable contract", ensuring continued upgradability. This removes

ambiguity due to function selector clashes since the admin functionality and the implementation logic are stored in the same contract

. As the Solidity compiler cannot handle function selector clashes across different smart contracts but can reject calls that clash across the same smart contract, function selector clashes are mitigated.

UUPS Pros and Cons

UUPS proxies are smaller, since the admin logic is stored in the implementation contract, and are therefore cheaper to deploy. This comes with the downside that deploying the implementation contract and therefore upgrading the contract, is more costly.

The other major upside but also drawback is that upgrades can be frozen by upgrading the proxy contract to an implementation that does not inherit a “proxiable contract”. However, if this is done in error, this is irreversible and the proxy cannot be upgraded in the future.

What is the Beacon Proxy Pattern?

The Beacon Proxy pattern introduces an efficient upgrade mechanism for when multiple proxy contracts require synchronized updates

, all referring to one implementation contract. To upgrade each of the proxies simultaneously, without having to update the implementation contract address in each proxy individually, each proxy instead points to an intermediate contract, known as “The Beacon” contract, which holds the implementation address to be used by all associated proxies.

The owner of the beacon proxy performs an upgrade by modifying the address of the implementation contract. When the child proxies refer to the beacon for the implementation address, they will be pointed to the new implementation contract for storage and state modification.

Image: The Beacon Proxy Pattern

Example use cases include enforcing push upgrades to all users’ proxy contracts when critical updates are needed. This way, security is maintained for all users without relying on each user individually upgrading their contracts.

Beacon Proxy Downsides

The downside of beacon proxy contracts is that the beacon acts as a centralized point of failure. The owner of the beacon contract can push upgrades to all associated proxies without their permission. To mitigate this, mechanisms such as multi-signature wallets can be implemented.

Before we wrap up, two more proxy patterns are worth mentioning: the minimal proxy pattern and the diamond proxy pattern.

What is the Minimal Proxy Pattern?

Minimal proxies are distinct from the previously mentioned proxy standards as they do not provide upgrade or authorization functionality

. Instead, they provide a streamlined and gas-optimized solution for deploying multiple instances of the same contract that share common logic

but each requires individual and distinct storage.

The minimal proxy pattern allows one instance of the implementation contract to be deployed and then lightweight or “minimal” proxies to be deployed for each new contract instance. These proxies are static and immutable post-deployment, simplifying their structure and reducing deployment and runtime gas costs.

What is the Diamond Proxy Pattern?

The diamond pattern, allows proxy contracts to delegate function calls to multiple implementation contracts known as "facets". To implement the diamond pattern, a mapping in the proxy contract links function selectors to facet addresses. When a user initiates a function call, the proxy contract checks the mapping, identifies the corresponding facet, and uses a delegatecall

to redirect the call to the appropriate implementation contract.

This upgrade pattern offers several advantages compared to other proxy upgrade patterns:

1. It enables single functions to be modified without requiring the entire implementation contract to be redeployed.
2. The diamond pattern allows functions to be split across multiple implementation contracts meaning that the 24KB smart contract size limit no longer applies.
3. Unlike proxy patterns with broad access control, where accounts with certain permissions can upgrade the entire contract

, the diamond pattern supports modular permissions. This allows restrictions to be set to only be able to upgrade specific functions within the smart contract.

Image: The Diamond Proxy Pattern

Upgradable Solidity smart contracts Proxy Patterns comparison

Now that we've learned about the different proxy patterns for upgrading smart contracts, let's compare them to better understand their advantages and disadvantages:

Proxy Pattern Advantages Disadvantages Simple/Delegate - Upgrade and authorization functionality.

Security Considerations

As well as the aforementioned advantages and disadvantages of the different proxy patterns it is also important to note the following security considerations when deciding whether to write upgradable smart contracts:

- Storage and Function Collisions

: as described above, storage and function collisions can pose a security risk when using proxy contracts. This was worth mentioning again since it can so often be an issue with the security of a smart contract.

- Added complexity

: proxy patterns provide an extra layer of complexity to smart contract design. With each upgrade the complexity, and therefore the difficulty in understanding the smart contract increases. This makes the smart contract or protocol harder to audit and therefore can make it more vulnerable to attack. To mitigate this, make sure that all upgrades are properly documented and upgrades are minimized.

- Centralization

: the action of performing an upgrade is centralized as it requires trust in the authorized address to perform the upgrade. Since the logic executed can be arbitrarily modified, users can be vulnerable to attack via malicious admins. To mitigate this, use multi-signature accounts to perform admin actions.

Summary

Proxy smart contract patterns are effective ways to create upgradable smart contracts meaning that the implementation logic can be modified while maintaining storage and address immutability.

This means that bugs can be fixed and functionality modified post-deployment. There are different types of proxies depending on the specific use case, all with their advantages and disadvantages.

When deciding whether to use proxy contracts, it is important to consider the security considerations of these patterns and properly document these decisions and implications for users.